6-20-05 Viewed not same

XML for Data: Reuse it or lose it, Part 3

PDF e-mail it!

Realize the benefits of reuse

Level: Intermediate

Kevin Williams (kevin@blueoxide.com)
CEO, Blue Oxide Technologies, LLC
08 Jul 2003

In the final installment of this three-part column, Kevin Williams looks at some of the ways you can take advantage of the reusable XML components that he defined in the previous two installments of this column. Designing XML with reusable components can, in many ways, create direct and indirect benefits; Kevin takes a quick look at some of the most important.

Contents:

Reusable XSLT components
Class-to-XML mapping (fragment serialization, deserialization)
Bringing XML and Web services together
Conclusion
Resources
About the author
Rate this article

Related content:

Reuse it or lose it, Part 1
Reuse it or lose it, Part 2

Subscriptions:

dW newsletters
dW Subscription (CDs and downloads)

This column builds on the philosophy of XML reuse I described in the first two columns, so if you haven't read those yet you might want to before diving into this one (see Resources).

The first benefit of using reusable components isn't necessarily a direct benefit of the design of XML structures that use components, but it is a natural outcome of the approach. To create components that can be reused, you need to capture solid semantics about those components. These semantics can be extended into the processing code itself to make the programmer's job easier. Take a look at the brief example in Listing 1. Suppose you have the following customer XML document:

Listing 1. Example customer XML document

```
<customer>
  <name>Amalgamated Widgets, Inc.</name>
  <contact>Fred Smith</contact>
  <phone>304-555-1212</phone>
</customer>
```

If you were to design this document as a single XML schema, you might not capture the semantics for each datapoint in the schema -- which would make it more difficult to write code to accurately process instances. (For example, is `contact` a name or an e-mail address? Is the order of the name *first name, middle name, last name* or *last name, comma, first name, middle name*?) On the other hand, if you choose to design this document using reusable XML components (datapoints for `name`, `contact`, and so on), you have already forced yourself to capture good semantics, because you can't reuse the components without knowing precisely the semantics of those components. This means that the processing software can take these semantic constraints as read and simplify the programmer's job.

Reusable XSLT components

Another natural benefit of the component-based approach to XML design is the ability to reuse XSLT fragments to ensure a standardized presentation of information across many different documents. Again, this is a natural outcome of capturing good semantics and reusing elements and attributes whenever possible. Suppose you have the following two documents:

Listing 2. Example customer and supplier XML documents with no reuse

Microsoft.com Home | Site Map

6-20-05 Viewed not same

msdn

MSDN Home | Developer Centers | Library | Downloads | Code Center | Subscriptions | MSDN Worldwide

Search for

MSDN Magazine Go

Advanced Search

MSDN Home > MSJ

MSJ Home

Search

Source Code

Back Issues

Subscribe

Reader Services

Write to Us

MSDN Magazine

MIND Archive

Magazine Newsgroup

February 1996

MICROSOFT SYSTEMS JOURNAL

Programming Windows 95 with MFC, Part VII: The Document/View Architecture

Jeff Prosise

Jeff Prosise writes extensively about programming in Windows and is a contributing editor of several computer magazines. He is currently working on a book, Programming Windows 95 with MFC, *to be published this winter by Microsoft Press.*

Click to open or copy the LIFE project files.

In the early days of MFC, applications were built very much like the sample programs in Parts I through VI of this series. An application had two principal components: an application object representing the application itself, and a window object representing the application's window. The application object's primary duty was to create the window object, and the window object, in turn, processed messages. Other than the provision of general-purpose classes such as CString and CTime to represent non-Windows objects, MFC was little more than a thin wrapper around the Windows API. It grafted an object-oriented interface onto windows, dialog boxes, device contexts, and other objects already present in Windows® in one form or another.

MFC 2.0 changed the way that Windows-based applications are written by introducing the document/view architecture. This architecture is carried through to MFC 4.0. In a document/view application, an application's data is represented by a document object and views of that data are represented by one or more view objects. The document and view objects work together to process the user's input and draw textual and graphical representations of the resulting data. MFC's CDocument class serves as the base class for all document objects, while the CView class and its derivatives serve as base classes for view objects. The top-level window, which is derived from either CFrameWnd or CMDIFrameWnd, is no longer the focal point for message processing, but serves primarily as a container for views, toolbars, status bars, and other objects.

The document/view architecture simplifies the development process. Code to perform routine chores such as prompting the user to save unsaved data before a document is closed is provided for you by the framework. So is code to transform your application's documents into OLE containers, simplify printing, use splitter windows to divide a window into two or more panes, and more.

MFC supports two types of document/view applications. The first is single-document interface (SDI) applications, which support just one open document at a time. The second is multiple-document interface (MDI) applications, which permit the user to have two or more documents open concurrently. MDI apps support multiple views of each document, but SDI apps are generally limited to one view per document. Thanks to the support lent by the framework, writing an MDI application with MFC is only a little more work than writing an SDI application. But because the Windows Interface Guidelines for Software Design (Microsoft Press, 1995) and the online documentation that comes with Visual C++™ suggest you use SDI instead of MDI, this article, the

6-20-05 Viewed not same

Efficient Java RMI for Parallel Programming

JASON MAASSEN, ROB VAN NIEUWPOORT, RONALD VELDEMA,
HENRI BAL, THILO KIELMANN, CERIEL JACOBS, and RUTGER HOFMAN
Vrije Universiteit, Amsterdam

Java offers interesting opportunities for parallel computing. In particular, Java Remote Method Invocation (RMI) provides a flexible kind of remote procedure call (RPC) that supports polymorphism. Sun's RMI implementation achieves this kind of flexibility at the cost of a major runtime overhead. The goal of this article is to show that RMI can be implemented efficiently, while still supporting polymorphism and allowing interoperability with Java Virtual Machines (JVMs). We study a new approach for implementing RMI, using a compiler-based Java system called Manta. Manta uses a native (static) compiler instead of a just-in-time compiler. To implement RMI efficiently, Manta exploits compile-time type information for generating specialized serializers. Also, it uses an efficient RMI protocol and fast low-level communication protocols.

A difficult problem with this approach is how to support polymorphism and interoperability. One of the consequences of polymorphism is that an RMI implementation must be able to download remote classes into an application during runtime. Manta solves this problem by using a dynamic bytecode compiler, which is capable of compiling and linking bytecode into a running application. To allow interoperability with JVMs, Manta also implements the Sun RMI protocol (i.e., the standard RMI protocol), in addition to its own protocol.

We evaluate the performance of Manta using benchmarks and applications that run on a 32-node Myrinet cluster. The time for a null-RMI (without parameters or a return value) of Manta is 35 times lower than for the Sun JDK 1.2, and only slightly higher than for a C-based RPC protocol. This high performance is accomplished by pushing almost all of the runtime overhead of RMI to compile time. We study the performance differences between the Manta and the Sun RMI protocols in detail. The poor performance of the Sun RMI protocol is in part due to an inefficient implementation of the protocol. To allow a fair comparison, we compiled the applications and the Sun RMI protocol with the native Manta compiler. The results show that Manta's null-RMI latency is still eight times lower than for the compiled Sun RMI protocol and that Manta's efficient RMI protocol results in 1.8 to 3.4 times higher speedups for four out of six applications.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*; D.3.2. [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages; object-oriented languages*; D.3.4 [**Programming Languages**]: Processors—*compilers; run-time environments*

General Terms: Languages, Performance

Additional Key Words and Phrases: Communication, performance, remote method invocation

Authors' address: Division of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2001 ACM 0164-0925/01/1100–0747 $5.00